# "Crystallizing" FORTRAN

*Dong-Yuan Chen*[1]
*Marina C. Chen*
Department of Computer Science
Yale University
New Haven, CT 06520

*Ron Y. Pinter*
IBM Scientific Center
Technion City
Haifa 32000, Israel

*Shlomit S. Pinter*
*Dept. of Electrical Engineering*
Technion — Israel Institute of Technology
Haifa 32000, Israel



November 26, 1991

**Abstract**

The parallelization techniques embodied in source to source tools (that are customarily applied to FORTRAN programs) and in compilers for high level functional languages (such as Crystal) can and should be combined under a common framework. Traditionally, the former tools target shared memory machines, whereas the latter — distributed memory (or data parallel) machines. One effect of the combination is the ability to run sequential FORTRAN programs on a whole new class of machines; it also provides a good way to separate between different parallelization concerns and deal with them at the appropriate level.

We describe the design and an implementation of a prototypical system that enables such a combination. The design is modular and extendible; its implementation entails, among other challenges, bridging the gap between imperative and functional program semantics. We have developed several techniques that solve these and other fundamental problems, and present them within this concrete framework. Experimental results, as achieved by the prototype and as presented hereby, are most encouraging and substantiate the usefulness of this approach.



[1]**Corresponding Author:** Dong-Yuan Chen, 51 Prospect Street, New Haven, CT 06511, (203)432-4781, *chen-dong-yuan@cs.yale.edu*.

# 1 Introduction

Crystal is a very high level, functional language intended (mostly) for scientific programming of data parallel machines [4]. Several novel compilation techniques [6, 12, 13, 14] were developed for mapping Crystal programs to machines that support (especially) distributed memory in various ways. Prototype compilers for the Intel iPSC/2 and Thinking Machine's CM-2 that employ these techniques have been built (and are being further developed), enabling the effective and convenient programming of numeric algorithms for such machines.

The scientific-engineering community, however, still uses FORTRAN as its most common language (both for existing and new code). There is a large body of work [1, 2, 15, 16, 17] on how to analyze and subsequently parallelize FORTRAN programs. Most efforts in this area were directed at vector and shared-memory type machines, but also some work for distributed-memory machines has been emerging recently [3, 9, 19].

We argue that it would be beneficial to combine the parallelization and optimization technologies of the FORTRAN compilers and preprocessors with those of the various Crystal back-ends so as to achieve a more powerful language processing capability. For example, a path going from ordinary FORTRAN 77 programs all the way to a wide variety of data parallel machines, including those supported by the Crystal compilers, will become available.

In this paper we present and justify the design of a system that realizes such a combination, discuss various issues of program representation and transformation that arise in such a design, describe the techniques that are required and used to implement it (with emphasis on the novel ones), and report some preliminary experimental results. The hard part is to bridge the gap between the imperative semantics that is inherent to FORTRAN programs and the functional style of Crystal which exposes the desired data parallelism.

From a broader perspective, this work addresses the formidable challenge of putting together several compilation techniques that are both different in nature and complementary in function into one framework. In particular, most parallelization techniques as applied to FORTRAN programs are performed at a source to source level, whereas the Crystal optimizations are (as are most other compiler optimizations) conducted at a lower level of representation. Providing a common optimization platform for this purpose, even in its present prototypical form, may prove beneficial by applying it to other domains as well; our experience sheds some light on the issue of how high level and low level optimizations and program representations can be combined.

We have demonstrated our design and techniques by implementing a prototype system that uses Parascope [11] (from Rice University) to generate FORTRAN code with explicit parallel control structures, which is translated into Crystal by our method and then compiled by the experimental Crystal compiler for the iPSC/2. The running times of the resulting programs were only slightly longer than those of Crystal programs that were written for the same application (using the same algorithm and the same Crystal compiler). This indicates that our approach is indeed sound and could lead to a significant capability with further improvements.

The rest of this paper is organized as follows: Section 2 outlines the structure of the tool, discusses its merits, and lists the technical problems to be solved. Then, in Section 3, we present the FORTRAN

to Crystal translator along with the various novel transformation techniques and the intermediate representations that are used. Section 4 describes some extensions to the basic scheme, followed by a report on the implementation and experimental results in Section 5. We conclude with some topics for further research.

## 2 Method, Issues, and Assumptions

Our approach is illustrated in Figure 1. One first applies a parallelization tool, *e.g.* KAP, PTRAN, ParaScope, Parafrase, Tiny, or VAST-2, to obtain a new program in a parallel dialect of FORTRAN; the output of the parallelization tool also contains information (in some form) about the dependence relations between statements. The second step is the novel part: we have to supply a translator that converts programs in an imperative, relatively low level (even with the added parallel constructs) programming language, to a functional language such as Crystal (our actual target is a textual, Crystal-like intermediate representation). Both dependence information from the parallelization tool and the FORTRAN source code are used by the translator; our objective is to generate correct Crystal code that is amenable to effective optimization by the Crystal compiler. Finally, the translator's output is handed over to the appropriate Crystal compiler for one of the target machines.

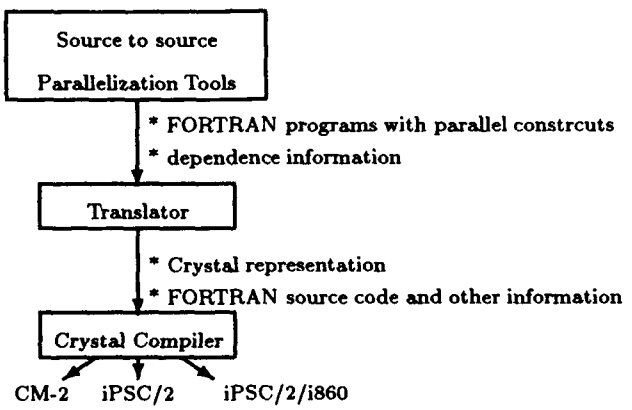


Figure 1: System Overview

**Major Advantages.** We see several advantages to this design. First, input programs can be written in FORTRAN 77 ("dusty decks" or new programs), but they can also be written in a parallel dialect (and then the process starts with the second stage). In the latter case, the program might already include the programmer's insights on, say, how to distribute an array, which will be passed on and translated to the Crystal intermediate representations. The normal feature in the Crystal compiler for automatically determining data distribution is bypassed.

Second, this design is both modular and portable in the sense that the intermediate forms are source programs (a parallel dialect of FORTRAN and a textual Crystal-like representation). Thus both the parallelizing tool (the "front-end") and the Crystal compiler (the "back-end") can be replaced by other than the original tools if deemed necessary. Furthermore, additional parallelization techniques,

such as the extraction of idioms [16], can be applied to the intermediate form, allowing further exploitation of Crystal constructs, such as scan and reduction.

Third, this design separates the shared memory from the distributed memory considerations. In the front-end we operate in a shared memory model; all the data distribution issues, such as communication synthesis, are deferred to the Crystal back-end.

**Key Issues.** Two key issues must be addressed to realize this design. First, what should the intermediate Crystal representation look like? How does this form preserve only the information that is essential to the correctness of the original programs without imposing unnecessary constraints on the execution order? Our design was influenced by the way control and data dependence relations of the sequential programs are maintained by the various parallelizing tools.

Second, how to translate imperative programs in multiple assignment form to a functional representation in single assignment form? In order to bridge this gap we have to (at least) eliminate output- and anti-dependences (per [20]). The single static assignment (SSA) form [8] can be used to represent the target program, but since the form as presented in the literature does not support arrays it must be generalized.

Finally, to simplify the initial design we made several assumptions that allowed us to concentrate on the main issues, as follows:

**Assumptions.** The input FORTRAN programs include only assignment *statements*, `DO` statements, and `IF` statements[1]. There are no subroutine calls in the programs; function calls are allowed, but functions must have no side effects. We also assume that the loop bounds of `DO` statements are constants and the strides have been normalized to 1 in all `DO` loops. The array references on the left hand side of assignment statements are assumed to have indices of the form `I + C` where `I` is a scalar variable and `C` is a constant. Some of these assumptions could be lifted by employing auxiliary data structures or by fine-tuning the translation algorithm.

We produce only interval domains (and Cartesian products thereof) in Crystal, thus focusing on rectangular arrays which are the only data structure available in FORTRAN. Also, we generate only "pure" Crystal code, *i.e.* there is no attempt to produce meta-language constructs that contain explicit parallelization directives.

We ignore the fact that those portions of the Crystal code that reside on the same node of the target machine must be translated back (by the compiler) into an efficient sequential program, so it might be advantageous to keep around the original FORTRAN code. This, again, could be remedied by using some auxiliary structures, or — alternatively — by encapsulating some designated inner loops as functions (after strip mining) and leaving them untranslated. This topic and the issues it raises (parameter passing etc.), however, are beyond the scope of this paper.

[1] For programs that are well structured the preprocessor can eliminate all `GOTO`s and turn them into structured constructs.

# 3 The Translator

As indicated above, the crux of the matter is to provide a translator from (a parallel dialect of) FORTRAN to Crystal. In this section we describe the intermediate Crystal representation to which we translate, present the translation algorithm, and demonstrate it on a simple example.

## 3.1 The Form of the Intermediate Crystal Representation

The intermediate Crystal representation to which we translate a FORTRAN program is a subset of the Crystal language augmented with some auxiliary data structures. Its intention is to capture both the control and data dependence relations that are present in FORTRAN programs in a uniform form. Thus all FORTRAN variable definitions are translated into Crystal data fields [10], which are (in this context) functionally defined arrays. Auxiliary data structures are used to keep information about expanded indices, formal index mappings, subroutines, etc.

In general, each data field definition has the form

$$a(I_1, ..., I_n) : D = \left\{ \begin{array}{l} p_1 \longrightarrow \tau_1 \\ \vdots \\ p_m \longrightarrow \tau_m \end{array} \right\}$$

where the $p_i$'s are Boolean expressions called *guards*, the $\tau_i$'s are arbitrary expressions, and $D = D_1 \times ... \times D_n$ is an $n$-dimensional domain which is the Cartesian product of interval domains $D_1, D_2, ..., D_n$. $(I_1, ..., I_n)$ is called the formal index of $a$; each $I_i$ ranges over the interval domain $D_i$. The value of $a(i_1, ..., i_n)$ is the value of expression $\tau_k$ where $k$ is the smallest integer such that $p_k$ is true (and undefined if all are false). We can imagine $a$ as an $n$-dimensional array defined over the domain $D$ whose value is defined by the conditional expression on the right hand side.

The intermediate representation has some basic properties: it is in single assignment form, each variable in the representation is defined only once, and the execution order is implicitly imposed by the data dependence relations between variable definitions. To demonstrate this, consider the following FORTRAN program segment.

```
S1:      do i = 1, 10
S2:         a = 2 * b + c
S3:         if ( i > 5 ) then
S4:            b = a + 1
S5:         endif
S6:      enddo
```

Here S2 is control dependent on S1, and S4 is control dependent on both S1 and S3. There is also an anti-dependence from S2 to S4, flow-dependences from S2 to S4 and from S4 to S2, and output-dependences from S2 to S2 and from S4 to S4. The corresponding Crystal representation (as produced by our translation algorithm) would be

$$a(i) : [1..10] = \{1 \leq i \leq 10 \longrightarrow 2 \times b(i-1) + c\}$$
$$b(i) : [1..10] = \left\{1 \leq i \leq 10 \longrightarrow \left\{ \begin{array}{l} i > 5 \longrightarrow a(i) + 1 \\ else \longrightarrow b(i-1) \end{array} \right\}\right\}$$

The control dependence relations in the FORTRAN program appear in the Crystal representation as the guards of data field definitions. The flow-, anti- and output-dependence relations are transformed into flow-dependence relations in the Crystal representation. Note that the scalar variables `a` and `b` in the FORTRAN program are transformed into one dimensional arrays in the Crystal representation; this is, in essence, scalar expansion along the time index. Since we know which dimension is expanded by the translation process, this information is passed on to the Crystal compiler using an auxiliary structure so that the original dimension could be restored during the code generation stage (if necessary).

The structure of our representation imposes some limitations on the translation process (as indicated in the previous section). For example, not all of the valid array references on the left hand sides of assignment statements can be translated into the formal index form in the Crystal representation. Some of them could be translated but would greatly increase the complexity of the translation algorithm and the complexity of the resulting representations. Also, `GOTO` statements, subroutine calls, and `COMMON` statements have no direct translation in the Crystal representation. Some of these limitations could be compensated for by using auxiliary data structures (see Section 4).

## 3.2 The Translation Algorithm

The translation algorithm divides the given parallel FORTRAN program into *segments* and makes each correspond to a single time step along a newly created time index in the Crystal representation. Then, as long as we can translate the variables defined in each segment into a single-assigned Crystal representation, multiple definitions of a variable in different segments turn into single-assignment form by the use of the new time index. We call this process time-dimension expansion because it expands the original time dimension of each variable by one to turn it into single-definition form. Thus, (a) will be translated into (b) in the following example.

```
x = exp1  (in program segment 1)
...
x = exp2  (in program segment 2)
...
```

(a)

```
x(t) : [1,T] = { if t == 1 then exp1
               || t == 2 then exp2
               || ...
             }
```

(b)

There are two main questions: first, how to divide the program into segments so that the statements in each segment can be translated into a corresponding Crystal representation in which each variable is defined only once, and — second — how do we translate the resulting segments as above. Since our algorithm follows the preprocessing stage, we can assume that the `DO` loops of the given program have been classified into sequential loops and parallel loops[2]. Then, the dividing lines between

[2]Recall that a `DO` loop can be parallel if there are no loop carried data dependences imposed by the loop between statements inside the loop body; otherwise it is a sequential loop.

segments are the start and end points of sequential loops. Specifically, we partition the program hierarchically into segments at the DO statements[3] and at the corresponding ends of the do blocks. Thus a *basic* segment contains at the top level only assignment, IF, and DOALL statements, but no DO statements. All other segments are *compound*.

To address the second question, we use three techniques for generating single-assignment form. A multiple definition of a variable resulting from a loop carried output dependence is resolved by *time index expansion* (which is similar to scalar expansion). Thus, such a variable will have at least as many time index dimensions as its nesting depth. Other multiple definitions in a segment are resolved by *controlled substitution* (a technique to be explained shortly) which takes care of anti dependence relations and branching execution, when possible. Lastly, *merging* definitions between segments (when substitution is impossible) is performed by a renaming-like approach which is done by adding another time index dimension. Note that within a basic segment we use only controlled substitution and merging; these techniques will be exemplified in what follows.

Controlled substitution prevents illegal forward substitution in the presence of anti dependences and branching executions. Let $S_i$ be a statement on which there exists at least one output dependent statement; then a controlled substitution is performed on every statement $S_j$ which is flow dependent on $S_i$. In order to match the correct values, two tags — [new] and [old] — are used by the substitution mechanism.

We demonstrate the technique by a brief example. It is important to note that we refer here only to non-loop carried dependences. In this example, x is multiply defined in (a) and with the tags the program (conceptually) looks like (b). Applying the substitution on x, following the flow dependences from S1 to S3 and from S1 to S4, we get the intermediate Crystal representation in (c). Note that since there is no flow dependence on z (from S2 to S3 or S4) no substitution need be performed on z. Finally, this program segment is merged with the preceding segment with the time index technique, resulting in the representation shown in (d).

```
S1:   x = exp1(z)
S2:   z = 5
S3:   y = x * c
S4:   x = x + exp2
```

(a) Original Program.

```
S1:   x[new] = exp1(z[old])
S2:   z[new] = 5
S3:   y[new] = x[old] * c[old]
S4:   x[new] = x[old] + exp2
```

(b) Program with variable tags.

```
z[new] = 5
y[new] = exp1(z[old]) * c[old]
x[new] = exp1(z[old]) + exp2
```

(c) Intermediate Crystal representation.

```
z(t) : [1,2] =  { if t == 1 then old value
                  || t == 2 then 5 }
y[new] = exp1(z(1)) * c[old]
x[new] = exp1(z(1)) + exp2
```

(d) After the merge on z.

The translation algorithm comprises two procedures that call each other recursively: the main entry point, Fortran_to_Crystal, handles compound segments, and the other — basic segments. During the translation, each program segment becomes a set of data field definitions in the Crystal representation where each variable that is defined in the program segment corresponds to a data

[3]From now on, we use DOALL to refer to a parallel loop and DO to refer to a sequential loop.

field definition. For convenience of presentation, we call the set of data field definitions resulting from translating a segment or a portion thereof a *definition unit (DU)*. The algorithm is described as follows:

```
PROCEDURE Fortran_to_Crystal(program)
   DU := empty set;
   WHILE not end of program DO
      IF current statement is a do statement THEN
         DU1 := Fortran_to_Crystal(loop body of do statement);
         DU2 := Time_Dimension_Expansion(DU1, loop index, loop bounds);
      ELSE
         DU2 := Basic_Segment_Translation();
      ENDIF
      DU := Time_Dimension_Merge(DU, DU2);
   ENDWHILE;
   RETURN(DU);
END Fortran_to_Crystal;

PROCEDURE Basic_Segment_Translation()
   DU := empty set;

   WHILE (not end of program) AND
         (current statement is not a do statement) DO

      X := current statement;

      IF (X is an assignment statement) THEN
          DU1 := Assignment_Statement_Transformation(X);
      ELSE IF (X is an if statement) THEN
          DU2 := Fortran_to_Crystal(the then part of X);
          DU3 := Fortran_to_Crystal(the else part of X);
          DU1 := If_Statement_Transformation(DU2, DU3, if_predicate);
      ELSE  /* X must be a doall statement */
          DU2 := Fortran_to_Crystal(the loop body of X);
          DU1 := Doall_Statement_Transformation(DU2, loop index,
                                                loop bound);
      ENDIF

      DU := Merge_and_Substitute(DU, DU1);

   ENDWHILE

   RETURN(DU);
END Basic_Segment_Translation;
```

The algorithm proceeds incrementally. It first tries to extract and translate the next program segment into a DU; then the new DU is merged with the DU that has been accumulated thus far. Should there be multiple definitions of the same data field in the new DU and old DU, the time-dimension expansion technique mentioned above is applied to merge them into a single data field definition. This operation continues until the end of the program is reached. If the next statement is a **DO** statement, **Fortran_to_Crystal** is called recursively to translated the loop's body into a DU; otherwise, **Basic_Segment_Transformation** is called to translate the next basic segment into a DU.

**Basic_Segment_Translation** works by translating into a DU one statement of a basic segment at a time and merging it with the previously obtained DU. Multiple definitions of data fields in the existing DU and the new one are resolved by applying the techniques described above. This process

continues until it reaches the end of the program, the end of a loop's body, or a new **DO** statement. The statements which are in the **then** or **else** part[4] of an **IF** statement or in the loop body of a **DOALL** statement are treated as programs and are translated into DUs by calling **Fortran_to_Crystal** recursively. The function of **Assignment_Statement_Transformation**, **If_Statement_Transformation**, **Doall_Statement_Transformation**, and the two merge procedures, is illustrated by a simple example in the next section.

## 3.3 A Simple Example

Let us use an example to further illustrate how the translation algorithm works. Consider the following FORTRAN program with **doall** constructs:

```
 1  real u(32), v(32), psi(32)
 2  doall i = 1, 30
 3     u(i + 1) = (psi(i + 1) - psi(i)) / 100000
 4  enddo
 5  u(1) = u(32)
 6  do j = 1, 20
 7     if (j .gt. 10) then
 8        doall i = 1, 31
 9           u(i) = 0.5 * u(i + 1)
10        enddo
11     else
12        doall i = 1, 32
13           u(i) = v(i) + u(i)
14        enddo
15     endif
16  enddo
```

Lines 2-5 form a basic segment and lines 7-15 form a potentially compound segment. In the basic segment, the statement in line 3 is first translated into $DU_1$ which consists of one data field definition[5], as follows:

$$DU_1 : \left\{ \; u(S0) : [1..32] = \left\{ \begin{array}{l} S0 == i+1 \longrightarrow (psi(S0) - psi(S0-1))/100000 \\ else \longrightarrow u(S0) \end{array} \right. \right.$$

The array index on the left hand side of the assignment statement is translated into the formal index $S0$ in the data field definition. The domain of the formal index $S0$ is determined from the array declaration of $u$. All references to variable $i$ on the right hand side of the assignment statement have to be replaced by an expression using formal index $S0$. After the formal index transformation, variable $i$ appears only in the guard of the conditional expression in $DU_1$.

Next, $DU_1$ and the **DOALL** loop at line 2 are combined and translated into $DU_2$:

$$DU_2 : \left\{ \; u(S0) : [1..32] = \left\{ \begin{array}{l} 2 \leq S0 \leq 31 \longrightarrow (psi(S0) - psi(S0-1))/100000 \\ else \longrightarrow u(S0) \end{array} \right. \right.$$

Here the loop index $i$ of the **DOALL** loop has been transformed into an interval [1..30] which is defined by its loop bounds. Since the **DOALL** loop contains no loop carried data dependences, this transformation preserves the semantics of the original loop.

[4] We take some liberties with FORTRAN syntax here in order to simplify the discussion.

[5] The tags described in the previous section are not shown in the data field definition.

$DU_3$ is the representation of the statement at line 5:

$$DU_3 : \left\{ \begin{array}{l} u(S0) : [1..32] = \left\{ \begin{array}{l} S0 == 1 \longrightarrow u(32) \\ else \longrightarrow u(S0) \end{array} \right. \end{array} \right.$$

Since $DU_2$ and $DU_3$ are in a basic segment, forward substitution is applied to merge them into a single definition unit $DU_4$. Those instances of $u$ in $DU_2$ and $DU_3$ which refer to the data field $u$ defined in $DU_2$ are replaced by the data field definition of $u$ in $DU_2$. After substitution, the definition of $u$ in $DU_2$ can be removed to obtain the new $DU_4$:

$$DU_4 : \left\{ \begin{array}{l} u(S0) : [1..32] = \left\{ \begin{array}{l} S0 == 1 \rightarrow \left\{ \begin{array}{l} 2 \leq 32 \leq 31 \rightarrow (psi(32) - psi(31))/100000 \\ else \rightarrow u(32) \end{array} \right. \\ else \rightarrow \left\{ \begin{array}{l} 2 \leq S0 \leq 31 \rightarrow (psi(S0) - psi(S0-1))/100000 \\ else \rightarrow u(S0) \end{array} \right. \end{array} \right. \end{array} \right.$$

$DU_4$ could be further simplified by removing redundant guards and definitions. After the simplification, we get $DU_5$:

$$DU_5 : \left\{ \begin{array}{l} u(S0) : [1..32] = \left\{ \begin{array}{l} S0 == 1 \rightarrow u(32) \\ 2 \leq S0 \leq 31 \rightarrow (psi(S0) - psi(S0-1))/100000 \\ else \rightarrow u(S0) \end{array} \right. \end{array} \right.$$

By applying similar transformations, the segment in lines 7-15 is translated into $DU_6$. Note that the **if-then-else** statement has been translated into a conditional expression in $DU_6$. The predicate of the **IF** statement is now a guard in the conditional expression.

$$DU_6 : \left\{ \begin{array}{l} u(S0) : [1..32] = \left\{ \begin{array}{l} j > 10 \rightarrow \left\{ \begin{array}{l} S0 \leq 31 \rightarrow 0.5 * u(S0+1) \\ else \rightarrow u(S0) \end{array} \right. \\ else \rightarrow v(S0) + u(S0) \end{array} \right. \end{array} \right.$$

Since the **do** loop at line 6 is a sequential loop, scalar expansion is applied to each data field definition in $DU_6$. A time dimension with domain [1..20] is added to data field $u$ and all references to $u$ are changed accordingly to reflect the new time dimension. Using scalar expansion, $DU_6$ is transformed into $DU_7$.

$$DU_7 : \left\{ \begin{array}{l} u(S0,T0) : [1..32] \times [1..20] = \left\{ \begin{array}{l} T0 > 10 \rightarrow \left\{ \begin{array}{l} S0 \leq 31 \rightarrow 0.5 * u(S0+1, T0-1) \\ else \rightarrow u(S0, T0-1) \end{array} \right. \\ else \rightarrow v(S0) + u(S0, T0-1) \end{array} \right. \end{array} \right.$$

Finally, $DU_5$ and $DU_7$ are merged along the new time dimension. $DU_5$ is treated as a single time step and is merged with $DU_7$ which is treated as a time interval. The merging process involves scalar expansion of some data field and time domain shifting to join two disjoint time intervals into one time interval. $DU_8$ is the final Crystal representation of the given FORTRAN program:

$$DU_8 : \left\{ \begin{array}{l} u(S0,T0) : [1..32] \times [0..20] = \\ \quad \left\{ \begin{array}{l} T0 == 0 \rightarrow \left\{ \begin{array}{l} S0 == 1 \rightarrow u(32, T0-1) \\ 2 \leq S0 \leq 31 \rightarrow (psi(S0) - psi(S0-1))/100000 \\ else \rightarrow u(S0, T0-1) \end{array} \right. \\ T0 >= 1 \rightarrow \left\{ \begin{array}{l} T0 > 10 \rightarrow \left\{ \begin{array}{l} S0 \leq 31 \rightarrow 0.5 * u(S0+1, T0-1) \\ else \rightarrow u(S0, T0-1) \end{array} \right. \\ else \rightarrow v(S0) + u(S0, T0-1) \end{array} \right. \end{array} \right. \end{array} \right.$$

# 4 Extensions to the Basic Scheme

Some of the assumptions on the input FORTRAN programs can be relaxed by the use of auxiliary data structures and modifications to the translation algorithm. In this section, some possible extensions to the Crystal representation and to the algorithm itself are presented to improve the translation process and to deal with a wider class of programs.

**Simplification of the Intermediate Representation.** A more compact and clean intermediate representation both improves the performance of the translation algorithm and helps the Crystal compiler generate better code; thus it is desirable to simplify the data field definitions to eliminate redundancy. A simple evaluator is used in the translator to evaluate the guards of conditional expressions under a set of known constraints. The initial constraints are derived from the domains of formal indices; if the values of Boolean expressions can be determined under these constraints, the guards or the whole conditional expressions are reduced to simpler forms. When one conditional expression is nested within another, it is evaluated under the new constraint which is derived from the initial constraint and the Boolean expressions guarding the conditional expression.

Consider the following data field definition in (a). The initial constraint derived from the index domain of $u$ is $C_1 \equiv \ulcorner 1 \leq S \leq 32 \urcorner$. The Boolean expression $\ulcorner 1 \leq S \leq 10 \urcorner$ is evaluated under the initial constraint $C_1$. Since the result could be either true or false, the value of $\ulcorner 1 \leq S \leq 10 \urcorner$ cannot be determined under $C_1$. The evaluator then tries to evaluate the conditional expression under the guard $\ulcorner 1 \leq S \leq 10 \urcorner$. The new constraint $C_2$ is $C_1 \bigcap \ulcorner 1 \leq S \leq 10 \urcorner \equiv \ulcorner 1 \leq S \leq 10 \urcorner$. The Boolean expression $\ulcorner 1 \leq S \leq 31 \urcorner$ is found to be always true under $C_2$, therefore the whole conditional expression can be reduced to the value $a$. Similarly, the **else** part of the top level conditional expression is evaluated under the constraint $\ulcorner 11 \leq S \leq 32 \urcorner$ and redundant Boolean expressions removed. The data field definition after simplification is shown in (b).

$$u(S) : [1..32] = \begin{cases} 1 \leq S \leq 10 \rightarrow \begin{cases} 1 \leq S \leq 31 \rightarrow a \\ else \rightarrow b \end{cases} \\ else \rightarrow \begin{cases} 20 \leq S \leq 32 \rightarrow c \\ else \rightarrow d \end{cases} \end{cases}$$

(a)

$$u(S) : [1..32] = \begin{cases} 1 \leq S \leq 10 \rightarrow a \\ 20 \leq S \rightarrow c \\ else \rightarrow d \end{cases}$$

(b)

**Translation of Formal Indices.** In the translation process, the array indices on the right hand sides of assignment statements have to be translated into formal indices in data field definition. Consider the statement $\ulcorner a(i * j, i + j) = i - j \urcorner$. To translate it into Crystal, we have to solve the equations $\ulcorner S0 = i * j \urcorner$ and $\ulcorner S1 = i + j \urcorner$ such that $i$ and $j$ on the right hand side could be written as expressions of $S0$ and $S1$.

In general, solving equations to translate the array indices into the formal indices of data field is difficult and even impossible in some cases. Even when the solution can be found, it will result in complicated expressions which will hurt the performance of the generated parallel programs. Instead of solving these equations, auxiliary data structures could be used to capture the relation between

$i, j$ and $S0, S1$. These auxiliary data structures could be passed on to the Crystal compiler together with the Crystal representations such that the original assignment statement could be restored in the code generation stage of the Crystal compiler. Thus the statement $\ulcorner a(i*j, i+j) = i - j \urcorner$ could be translated into the following data field definition

$$a(S0, S1) : D = \begin{cases} (S0 == i*j) \textbf{ and } (S1 == i+j) \rightarrow \$fun.i - \$fun.j \\ else \rightarrow a(S0, S1) \end{cases}$$

where $\$fun = (S0 = i*j, S1 = i+j)$ is an auxiliary data structure. The translation algorithm has to be modified to deal with this kind of data field definition. This way, the restriction on array indices could be relaxed.

**Subroutine Calls.** Suppose that the parallelizing tools are able to extract the interprocedural data flow information in the preprocessing stage, and that the information about which variables are referenced and which variables are updated in the subroutines is available to the translator. We could then translate each subroutine call in FORTRAN into a single time step along some time dimension in Crystal. Consider the subroutine **ADD** in (a) which takes arrays $a$ and $b$, adds them together, and stores the result in array $a$. One possible translation into Crystal is shown in (b).

```
real a(100), b(100)
...
call ADD(a, b)
...
```

(a)

$$a(S, T) : D = \begin{cases} \vdots \\ (T == t') \rightarrow ADD(a(*, t-1), b(*)).a \\ \vdots \end{cases}$$

(b)

Properties associated with the subroutine **ADD**, *e.g.* variables referenced or updated in the subroutine and even the source code of the subroutine, could be stored in auxiliary data structures. However, the problems of how to integrate the subroutine information into the Crystal compiler and to generate correct parallel programs remain to be investigated.

# 5 Implementation and Results

A prototype of the FORTRAN to Crystal translator was implemented in T [18]. It by itself consists of a lexical analyzer, a parser, the translation module, a simplifier, and a code generator. We used Parascope [11] from Rice University as the source to source FORTRAN parallelization tool, and the Crystal compiler for the Intel iPSC/2, a hypercube based machine.

A weather prediction program, based on shallow-water equations and consisting of about 120 lines of FORTRAN code, was used to both test and benchmark the translator. It was processed successfully by our prototype translator and was also translated by hand, thereby producing two Crystal programs. These programs were compiled by a prototype Crystal compiler for the iPSC/2 (followed by some hand tuning, because the Crystal compiler is still under development). The exact same communication pattern was generated for the two programs, which is quite remarkable; the difference between the computation times (not including the communication time, which should be the same)

of the two programs was about 25% in favor of the hand written program. Further experimentation to compare the performance differences between auto-translated and hand-translated versions of this and other Crystal programs is underway. We are already witnessing a certain reduction of the above gap by enabling certain advanced optimizations in the Crystal compiler.

The automatically translated Crystal program was run both on the iPSC/2 at Yale and the iPSC/2/i860 at Oak Ridge National Labs. Table 1 shows the speedup and efficiency as a function of the number of processors when program size and the number of iterations are fixed. Note that the rather impressive speedups are largely due to the simple and well structured nature of the application program.

| Procs | Total(ms) | $T_{comp}$ | $T_{shift}$ | Mflops | Speedup | Eff. |
|---|---|---|---|---|---|---|
| 1 | 331289 | 331289 | 0 | 0.39 | 1.00 | 1.00 |
| 2 | 167953 | 164114 | 3924 | 0.76 | 1.97 | 0.98 |
| 4 | 83375 | 78454 | 4995 | 1.53 | 3.97 | 0.99 |
| 8 | 44208 | 39924 | 5286 | 2.89 | 7.49 | 0.94 |
| 16 | 24134 | 20308 | 4425 | 5.30 | 13.72 | 0.85 |
| 32 | 13208 | 10178 | 3319 | 9.68 | 25.08 | 0.78 |
| 64 | 7486 | 5314 | 2326 | 17.07 | 44.25 | 0.69 |

(a) The iPSC/2 at Yale. Problem size: 128x128, 120 iterations.

| Procs | Total(ms) | $T_{comp}$ | $T_{shift}$ | Mflops | Speedup | Eff. |
|---|---|---|---|---|---|---|
| 1 | 180516 | 180516 | 0 | 2.8 | 1.0 | 1.00 |
| 2 | 91715 | 90529 | 1225 | 5.5 | 1.9 | 0.95 |
| 4 | 47186 | 45537 | 1691 | 10.8 | 3.8 | 0.95 |
| 8 | 22049 | 20481 | 1630 | 23.2 | 8.2 | 1.03 |
| 16 | 11803 | 10369 | 1507 | 43.3 | 15.3 | 0.96 |
| 32 | 6583 | 5219 | 1426 | 77.7 | 27.4 | 0.86 |
| 64 | 3987 | 2684 | 1361 | 128.2 | 45.3 | 0.71 |

(b) The iPSC/2/i860 at Oak Ridge National Labs. Problem size: 256x256, 120 iterations.

Table 1: The speedup and efficiency using a different number of processors when problem size and the number of iterations are fixed. $T_{comp}$ is the computation time and $T_{shift}$ is the communication time plus the time for buffer copying (both measured in msec). The results for a single processor on the iPSC/2/i860 are extrapolated due to memory constraint.

# 6 Conclusions and Further Research

The approach of efficient compilation of sequential FORTRAN programs for data parallel machines using both shared memory parallelization and Crystal compilation techniques was proposed. Novel and interesting solutions to the variety of technical problems that arise with such a scheme were presented, and the feasibility was demonstrated with surprisingly good results (at least for a certain class of FORTRAN programs). Certainly, more experimentation is necessary to validate this further, and we hope to report the results in the full paper, if accepted.

Several issues remain to be investigated: In order to support a more general class of FORTRAN

programs, the intermediate Crystal representation should be augmented. The translation algorithm should be improved to generate efficient and compact Crystal programs. Extraction of program idioms like the scan and reduce operations could be added into the translator in the future to achieve better performance. Program transformation techniques on Crystal source programs and optimization techniques for the Crystal compiler also deserve further research effort to generate parallel code with more parallelism and higher performance.

On a more general level, can all the techniques that are involved in this process be unified under one cohesive framework, *i.e.* work on the same intermediate representation? Also, is such a "closed" design really better than our "open" approach? Finally, can some of these methods and techniques be applied to different settings of language pairs and target machines?

## Acknowledges

Generous support from the Office of Naval Research under Contract N00014-91-J-1559 and the National Science Foundation under Grant CCR-8908285 are gratefully acknowledged. We would like to thank C.-Y. Lin for conducting the experiment on the iPSC/2 and iPSC/2/i860.

## References

[1] F. E. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante. An overview of the PTRAN analysis system for multiprocessing. *Journal of Parallel and Distributed Computing*, 5(5):617–640, October 1988.

[2] R. Allen and K. Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Trans. on Programming Languages and Systems*, 9(4):491–542, October 1987.

[3] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. An interactive environment for data partitioning and distribution. In *Fifth Distributed Memory Computing Conference*, April 1990.

[4] M. Chen, Y.-I. Choo, and J. Li. Crystal: Theory and pragmatics of generating efficient parallel code. In *Parallel Functional Languages and Compilers*, chapter 7, pages 255–305. ACM Press and Addison-Wesley, 1991.

[5] M. C. Chen, Y.-I. Choo, and J. Li. Compiling parallel programs by optimizing performance. *The Journal of Supercomputing*, 2(2):171–207, October 1988.

[6] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *Sixteenth Annual Symposium on Principles of Programming Languages*, pages 25–35. ACM, January 1989.

[7] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C. Tseng. An overview of the Fortran D programming system. Technical Report TR91-154, Dept. of Computer Science, Rice University, March 1991.

[8] M. Jacquemin and J. A. Yang. Crystal reference manual version 3.0. Technical Report YALEU/DCS/TR-840, Dept. of Computer Science, Yale University, Jan. 1991.

[9] K. Kennedy, K. S. McKinley, and C.-W. Tseng. Interactive parallel programming using the parascope editor. *IEEE Trans. on Parallel and Distributed Systems*, 2(3):329–341, July 1991.

[10] J. Li and M. Chen. Compiling communication-efficient programs for massively parallel machines. *IEEE Transaction on Parallel and Distributed Systems*, July 1991.

[11] J. Li and M. Chen. The data alignment phase in compiling programs for distributed-memory machines. *Journal of Parallel and Distributed Computing*, October 1991.

[12] L.-C. Lu and M. C. Chen. Subdomain dependency test for massive parallelism. In *Supercomputing'90*, pages 962–972, November 1990.

[13] D. Padua and M. Wolfe. Advanced compiler optimizations for supercomputer. *Communications of the ACM*, 29(12):1184–1201, Dec. 1986.

[14] S. S. Pinter and R. Y. Pinter. Parallelizing programs using idioms. In *Eighteenth ACM Symposium on Principles of Programming Languages*, pages 79–92, January 1991.

[15] C. D. Polychronopoulos, D. J. Kuck, and D. A. Padua. Execution of parallel loops on parallel processor systems. In *International Conference on Parallel Processing*, pages 519–527. IEEE, 1986.

[16] J. A. Rees, N. I. Adams, and J. R. Meehan. *The T manual*, fifth edition, October 1988.

[17] P.-S. Tseng. *A Parallelizing Compiler for Distributed Memory Parallel Computers.* PhD thesis, Carnegie Mellon University, May 1989. CMU-CS-89-148.

[18] M. Wolfe. *Optimizing Supercompilers for Supercomputers.* MIT Press, Cambridge, MA, 1989.